Big data

Data science

Data science

Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from data in various forms, both structured and unstructured.

Data science is made possible by the increasing availability of large volumes of data. We usually refer to this as **big data**.

Some fields in which data science is applied are:

- **Recommender algorithms**, like the ones Netflix uses to suggest TV shows
- **Translation services** like Google Translate
- **Genomic data**, which finds correlation between genes and diseases
- **Medicine**, where lifestyle and environment variables can be monitored and analysed by smart devices

From a technical standpoint, in order to "perform" data science, a data scientist needs to perform two steps:

1. **Collect** all the data: the more, the better, since more samples bring statistical relevance. Furthermore, keeping all the collected data is much cheaper than actually cherrypicking
2. **Decide** what to do with the data. Usually, when a question arises, experiments are run on the data

This is very different from the usual information systems approach, where the decision on what data to keep is done immediately, when the data is collected. The consequences of this approach are:

- **Volume**: there is going to be a lot of data
- **Velocity**: data is going to arrive fast

- **Variety**: data will have different formats, different versions and possibly different sources
- **Veracity**: data will not be always correct

In order for data science to work, the tools that enable it must be able to scale up for increasing data volumes, they must support quickly changing data and require the following functions:

- Automatic parallelisation and distribution
- Fault-tolerance
- Status and monitoring tools
- A clean abstraction for programmers

MapReduce

Let's suppose we have a map, which is represented like a graph, where nodes are points of interest and edges are roads. We want to compute the shortest path from each point to the nearest gas station.

To solve this, we can compute the shortest path between each gas station and each node with well known algorithms, like Dijkstra's; then, for each node, we need to compare the distance computed from each gas station and keep only the minimum.

We can further optimise this solution discarding nodes that are too far away. For example, we can assume that the maximum distance between any node and the nearest gas station is 50 km and discard paths longer than that.

By analysing the problem, we discover that we made some wrong assumptions:

- We can easily access the entire input data
- We can easily store and update the state of the computation
- They fit in the disk of a single node

In general, some problems only appear "at scale", in data-intensive applications.

Reading input and storing the intermediate state of the computation can easily become the bottleneck. If input/output states do not fit on one machine, we need distributed solutions, which in turn need coordination and communication, which can become the bottleneck.

To solve this problem, we need to consider the scale of the problem and the computing infrastructure we have.

We will now assume that our computing infrastructure is a Google datacenter of the early 2000s.

We will now summarise the assumptions we're making below:

Assumption	**Description**
Scale of the problem	Terabytes of data which do not fit on disks or memory
Computing infrastructure	Cluster of "normal" computers with hundreds or thousands of node, but no dedicated hardware

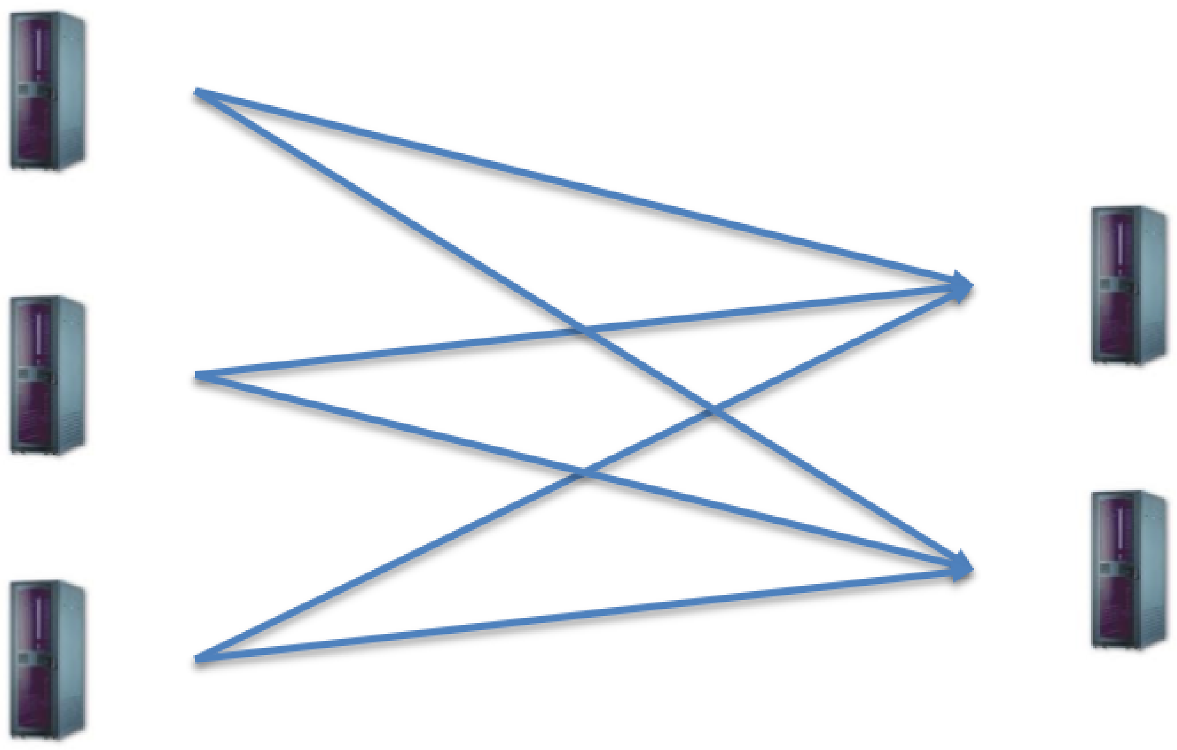
Assuming that the maximum distance between the point and the nearest gas station is 50 km, we can split the dataset into blocks b_i, each centred around gas station i. Different blocks can now be stored on different computers and, for each block b_i, we can compute the distance between gas station i and any node in the block.

Now, the computation is independent for each block and each block can start processing in parallel.

Once this is done, we need to determine the closest gas station and the shortest path for each node. This requires communication, since one point of interest can be at the intersection of several blocks. We can now repartition the data by node and the shortest path can be computed independently for each node.

We now have a solution that is bipartite:

- **Computation of paths**:
  - Data split by gas station
  - Each gas station can be processed in parallel
  - Nodes can read input blocks in parallel
- **Computation of shortest paths**:
  - Data split by node
  - Each node can be processed in parallel

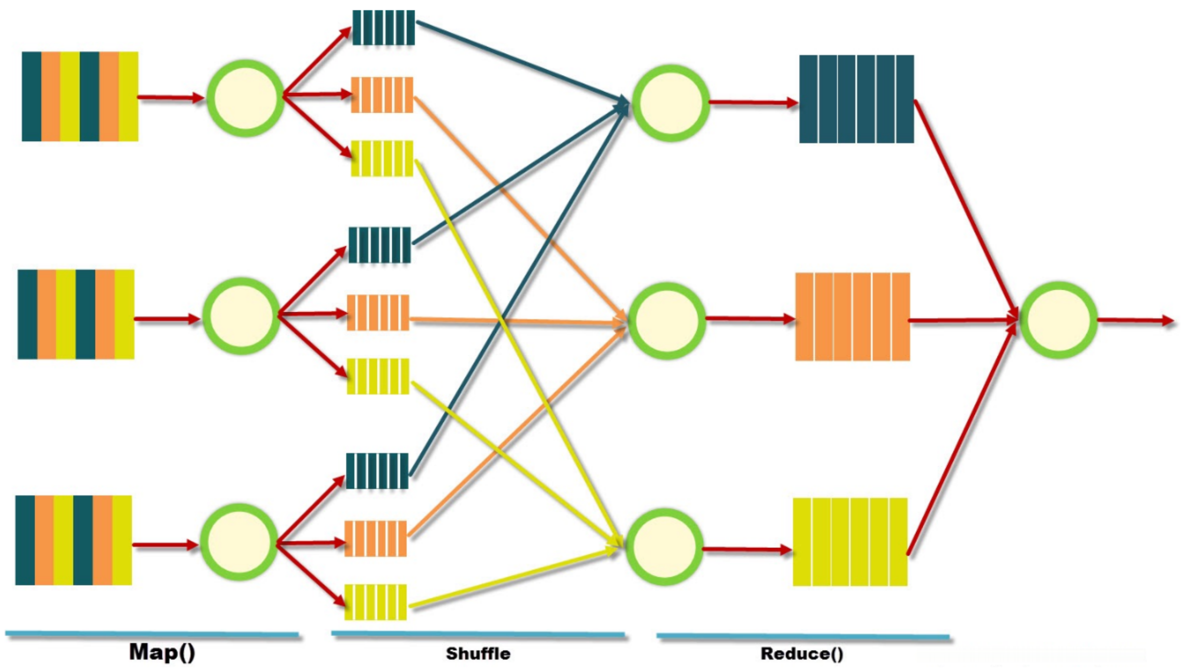


This is how the **MapReduce** programming model works. It was introduced by Google in 2004 and enables application programs to be written in terms of high-level operations on immutable data. The runtime system controls scheduling, load balancing, communication, fault tolerance, and more.

In MapReduce, computation is always split into two phases:

1. **Map**: processes individual elements and, for each one, outputs one or more key/value pairs
2. **Reduce**: processes all the values with the same key and outputs a value

The developers only need to specify the behaviour of these two functions.



In MapReduce, the platform manages:

- **Scheduling**: allocates resources for mappers and reducers
- **Data distribution**: moves data from mappers to reducers
- **Fault tolerance**: transparently handles the crash of one or more nodes

Scheduling

In MapReduce, input data is split into M map tasks, typically 64 MB in size, while the reduce phase is partitioned into R reduce tasks ($\text{hash}(k) \mod R$).

MapReduce is based on a master node and many worker nodes. Masters assign tasks to workers dynamically, considering the locality of the data of the assigned task.

Data locality

The goal of assigning tasks based on data locality is to limit the usage of network bandwidth. In GFS, data files are divided into 64 MB blocks and three copies of each are stored on different machines.

The master program schedules map tasks based on the location of these replicas: it executes map physically on the same machine as one of the input replicas.

This way, thousands of machines can read input at local disk speed.

Fault tolerance

Worker failure is detected by the master via periodic heartbeats. In case of a worker failure, both completed and in-progress map tasks on that worker should be re-executed, while only in-progress reduce tasks on that worker should do the same thing.

All reduce workers will be notified about any map re-execution.

If the master fails instead, the master's state is check-pointed to the global file system so that a new master can recover and continue from where the old one left off.

Stragglers

Stragglers are tasks that take a long time to execute due to slow hardware, poor partitioning, a bug or other causes.

In order to avoid stragglers, when a worker is done with most tasks, any remaining one is rescheduled. Furthermore, redundant executions are kept track of.

These two procedures significantly reduce overall run time.

Conclusions

MapReduce is very often used when, in order to solve a problem, a sequence of steps requires map and reduce operations to transform the input data and converge towards a single value answer.

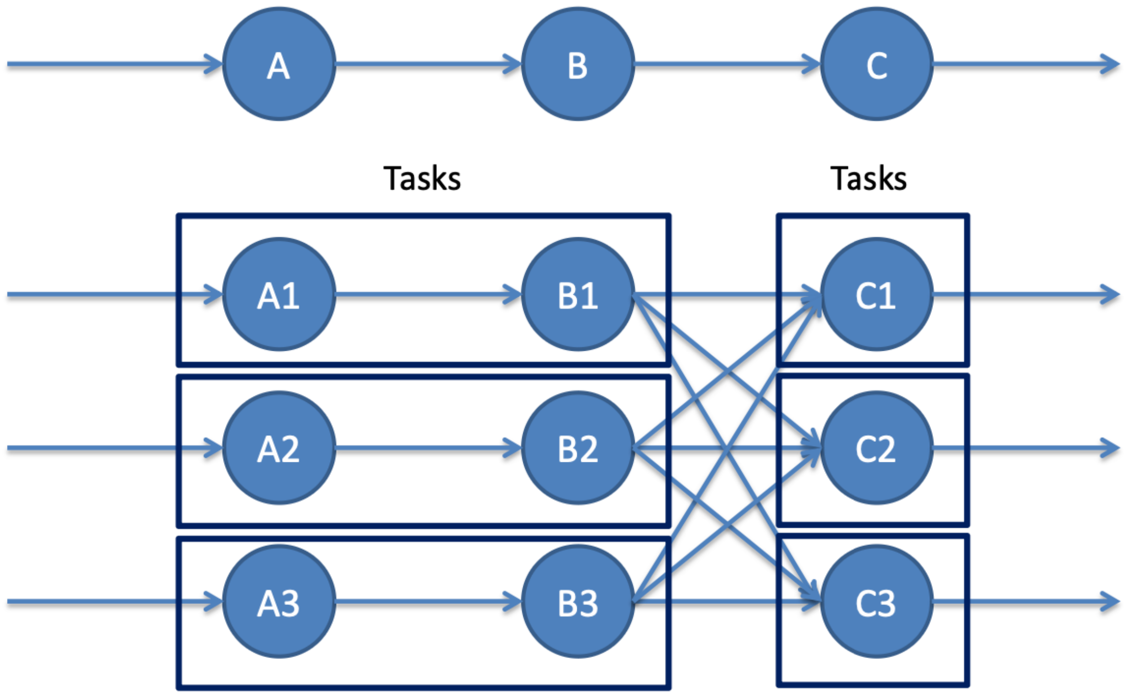
This method has both strengths and limitations:

Strengths	Limitations
The developers write simple functions	High overhead
The system manages complexities of allocation, synchronisation, communication, fault tolerance and stragglers	Lower raw performance than HPC
Very general	Very fixed paradigm
Good for large-scale data analysis	

Beyond MapReduce

In the last decade, many systems extended and improved the MapReduce abstraction in many ways:

- From two processing steps to arbitrary acyclic graphs for transformation, following the Dataflow model
- From batch processing to stream processing, which helps processing large datasets with high throughput and low latency
- From disk to main-memory or hybrid approaches



We will now consider two systems as representatives of two different architectural approaches, which differ in how or when they allocate tasks onto nodes:

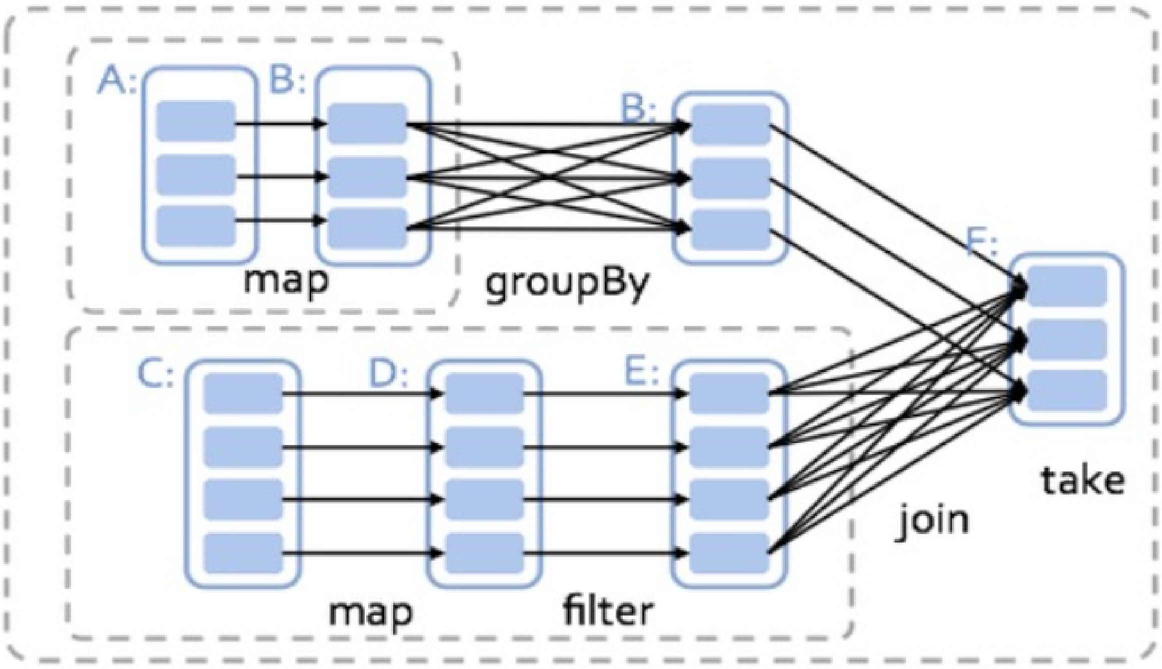
- Apache Spark: batch processing, scheduling of tasks
- Apache Flink: streaming or continuous processing, pipelining of tasks

Apache Spark

Apache Spark is similar to MapReduce, but instead of having only two stages, it can have as many as needed.

Intermediate results can be cached in main memory if they are reused multiple times.

Furthermore, the scheduling of tasks (or stages) ensures that the computation takes place close to the data.



Apache Spark supports streaming data through the micro-batch approach, in which the input data stream is split into small batches of data which are processed independently. Some state, however, can persist across batches and is stored as additional output for the next micro-batch.

Apache Flink

In Apache Flink, a job is not split into stages that are scheduled. Instead, all the operators are instantiated as soon as the job is submitted.

Operators communicate via TCP channels and an operator can start processing as soon as it has some data available from te previous ones. This is known as a **pipeline** architecture.

Apache Flink is ideal for stream processing, since the data flows into the system without waiting for the operators to be scheduled. This allows for lower latency.

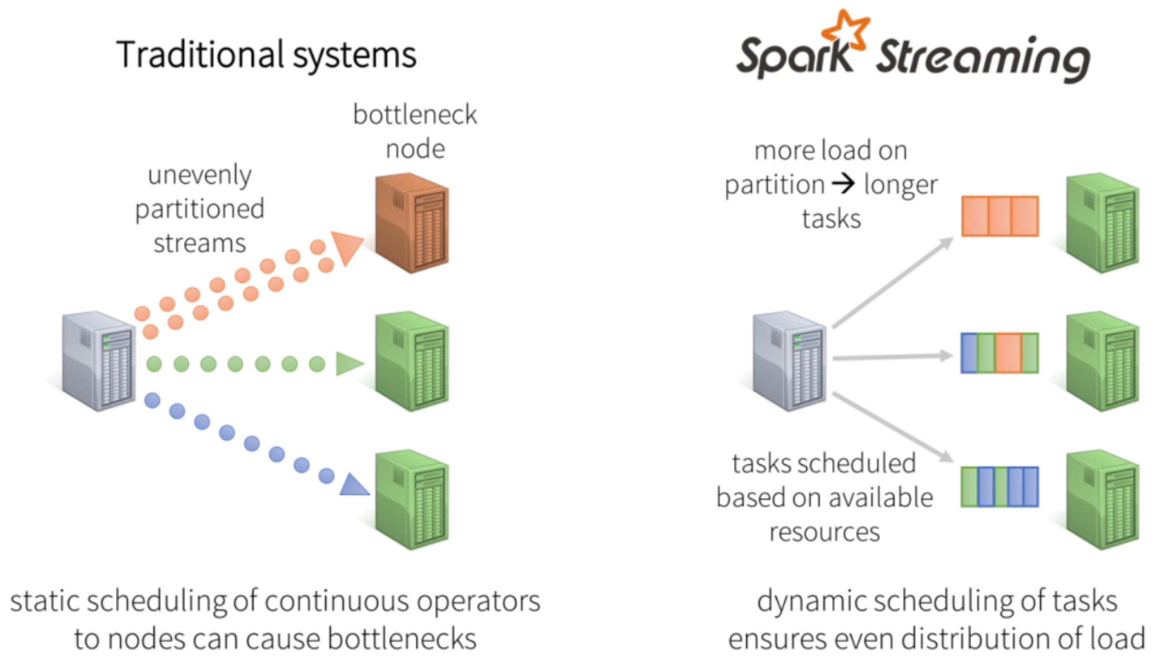
The same approach is also used for batch processing: the entire batch is streamed through the operators.

Comparison: Spark vs. Flink

The pipeline approach in Flink provides **lower latency**, which is relevant for stream processing scenarios where new data is continuously being generated. New data elements are ingested into the network of processing operators as they become available, so there is no need to accumulate batches or to schedule tasks.

However, a scheduling approach offers more opportunities to **optimise throughput**: moving larger data blocks can be more efficient, since it implies a smaller overhead from the network protocols and more opportunities for data compression.

Furthermore, scheduling decisions can consider data distribution: the scheduling approach of Spark **simplifies load balancing**, since dynamic scheduling decisions can consider data distribution, which is not possible in the case of a pipelined approach. This is also relevant for streaming workloads, where the distribution of data may change overtime.


Traditional systems
bottleneck node
unevenly partitioned streams
static scheduling of continuous operators to nodes can cause bottlenecks
Spark Streaming
more load on partition → longer tasks
tasks scheduled based on available resources
dynamic scheduling of tasks ensures even distribution of load

Scheduling approaches are also better for elasticity, since scheduling decisions take place dynamically at runtime. Providing elasticity is simply impossible for pipelined approaches.

As far as fault tolerance is concerned, if a node fails in Spark, since it offers no replication of intermediate results, it needs to recompute everything that's needed for further analysis.

In pipelined processing, instead, the whole system is periodically checkpointed and, in case of failures, everything can recover from the last checkpoint. Flink relies on this mechanism to ensure fault tolerance.